Analyse, Conception et Programmation par objets

Instructions pour le projet

Norbert Kajler et Fabien Moutarde
Norbert.Kajler@mines-paristech.fr, Fabien.Moutarde@mines-paristech.fr

60 Bd. Saint Michel 75272 Paris Cedex 06

Année 2011-2012
(Dernière mise à jour le 21 septembre 2011)

Table des matières

1 Principe et échéancier **2**

2 Avant-projet **3**

3 Projet **4**

4 Liste des projets **5**
4.1 Log and Profile (N. Kajler) ★★ 6
4.2 Carrelages (N. Kajler) ★★★ à ★★★★ 8
4.3 Logique « floue » (F. Moutarde) ★★★ à ★★★★ 10
4.4 Support Vector Machines (SVM) (F. Moutarde) ★★ à ★★★ 13

1 Principe et échéancier

Le contrôle du cours « Analyse, Conception et Programmation par objets » consiste à réaliser un projet **par groupes de 2 à 3 élèves** L'échéancier est le suivant :

- Constitution des groupes et choix initial des projets : **d'ici au lundi 26 septembre**
  (par e-mail à *Norbert.Kajler@mines-paristech.fr)*
- Séances d'assistance avant-projets : **après-midi des 29/9, 3/10 et 10/10**
- **Remise des avant-projets par les élèves : jeudi 13 octobre**
- Séance d'assistance projets : **après-midi du lundi 7 novembre**
- **Remise des projets par les élèves : jeudi 15 décembre**
- Soutenances (orales) des projets : **en janvier**

ATTENTION : le projet ne consiste pas seulement à développer en Java un logiciel qui marche ; la mise en oeuvre préalable correcte d'une méthode d'analyse et de conception objet en « UML » est une part essentielle de l'objectif du projet.

L'appréciation du projet portera sur :

- le sujet choisi : la difficulté du sujet sera prise en compte ;
- le document d'analyse remis lors de l'avant-projet ;
- le projet :
  - **la qualité de l'analyse et de la conception par objets** mise en oeuvre ;
  - les documents d'analyse, de conception et de réalisation accompagnant le projet ;
  - la qualité du code Java ;
  - la démonstration du programme.
- la soutenance orale du projet.

Il est à noter qu'en ce qui concerne le logiciel proprement dit, l'esthétique ou le raffinement de son interface homme-machine éventuelle sont moins importants que son bon fonctionnement, la qualité de sa conception, et la lisibilité du code.

2 Avant-projet

Le but de l'avant-projet est de présenter l'avancement du projet après l'étape d'analyse, ce qui permet de vérifier que le problème est bien compris et abordé correctement. Le document d'avant-projet (volume ≈ 15 à 25 pages, **A REMETTRE AU PLUS TARD JEUDI 13 OCTOBRE**) devra comporter (en les faisant apparaître **clairement**) l'ensemble des éléments suivants :

1. *Cahier des charges*. Si le sujet est un de ceux de la liste qui suit, résumer l'objectif du logiciel, et préciser clairement les restrictions ou les ajouts que vous apportez par rapport à son descriptif initial. Sinon définir clairement le cahier des charges de votre sujet. **Cette partie définit le contrat sur lequel vous vous engagez. Il est donc important d'être clair et précis.**

2. *Analyse*. Description textuelle et graphique (en notation UML) des principaux objets identifiés à partir du cahier des charges. Elle devra ainsi comporter :

   (a) le **dictionnaire** des éléments modélisés ;

   (b) la définition des besoins de l'utilisateur (use cases) ;

   (c) la liste des principales classes identifiées et leur organisation en packages ;

   (d) pour chaque package :

   - le diagramme de classes faisant apparaître les héritages et associations entre classes ;
   - la description (attributs et opérations) et représentation détaillée de chaque classe ;
   - des diagrammes états-transitions (seulement quand ils sont indispensables à la compréhension du mode d'emploi d'une classe) ;

   (e) **des diagrammes de séquence** illustrant le fonctionnement et l'interaction des principales classes.

3. **Echéancier des réalisations et répartition des tâches au sein du groupe.** Préciser l'ordre dans lequel vous pensez développer, tester, puis intégrer les différentes classes, et comment vous pensez répartir les tâches de développement au sein du groupe.

4. **Préparation de la validation.** Définir aussi précisément que possible des exemples d'utilisation du futur programme, qui permettront de vérifier, à la fin, qu'il fonctionne correctement (typiquement, un jeu de données d'entrées pour lesquelles on connaît le résultat qui doit être obtenu).

Le document d'avant-projet (ou tout au moins la partie *Analyse*) **devra être généré sous l'outil Modelio**. Ce document ne sera pas jugé par sa taille, mais en fonction de sa clarté et de sa pertinence : le « client » (qui est sensé ne connaître initialement que le cahier des charges) doit être capable de comprendre la modélisation objet proposée, et de vérifier que celle-ci devrait effectivement permettre de réaliser les services qu'il attend. De plus ne pas oublier que cette étape doit permettre à la personne encadrant le projet de vous éviter éventuellement de faire fausse route AVANT la conception détaillée et la réalisation effective du logiciel. En particulier, attention aux points suivants :

- les diagrammes doivent être **lisibles** (si il y a trop d'éléments et de liens, il vaut mieux subdiviser en plusieurs diagrammes) ;
- tous les noms (de classes, méthodes, associations, ...) doivent être « parlants » ;
- ne pas se contenter de juxtaposer sans commentaire les graphiques générés par « Modelio » : il DOIT y avoir des commentaires/explications, sinon le document sera indigeste et incompréhensible.

Le document d'avant-projet sera relu et annoté par un des enseignants, puis rendu, les remarques faites devant naturellement être prises en compte pour la suite du travail. N'hésitez pas à demander des éclaircissements à l'enseignant qui aura relu votre avant-projet si vous ne comprenez pas ses annotations.

3 Projet

Les documents écrits à remettre sont les suivants :

1. l'exemplaire du *Document d'analyse* remis lors de l'avant projet, et annoté manuellement par la personne encadrant le projet.
2. *Document de conception* : document similaire au document d'analyse, mais corrigé en fonctions des remarques faites sur l'avant-projet, et augmenté/modifié par tout ce qui aura été fait dans la phase de conception (regroupement de classes en packages, description complète et détaillées des classes, ajout des éventuelles classes « techniques », ...)
3. *Documentation technique du projet* :
   - Jeux de tests pour le projet complet.
   - Jeux de tests pour chacun des principaux objets que vous avez identifiés dans votre analyse et implantés : pour chaque objet, écrire un programme de test permettant de vérifier le fonctionnement de l'objet. **On ne peut que recommander l'élaboration de ces programmes au fur et à mesure de l'analyse et du développement.**
   - Listing **commenté** du projet (sur les machines Linux de l'Ecole on pourra obtenir un **listing bien lisible** en utilisant le script «`ccsi-print-source`»[1], ou la commande standard «`a2ps`»).
4. *Notice d'utilisation* qui permettra à vos « clients » :
   - d'installer votre logiciel ;
   - d'utiliser votre logiciel.

Par ailleurs, *l'ensemble (modèles UML, documents, code Java (source et exécutable))* devra être remis sous forme électronique dans un répertoire (par email ou clé USB). Un fichier **README** devra être inscrit également dans ce répertoire. Il devra comporter une description en quelques lignes de votre programme et de la façon de l'utiliser, ainsi qu'une présentation des différents fichiers présents dans le répertoire.

L'ENSEMBLE DU PROJET (TEL QUE DECRIT CI-DESSUS) DEVRA ETRE REMIS AU PLUS TARD LE JEUDI 15 DECEMBRE.

1. `ccsi-print-source` permet sur les machines Linux de l'Ecole d'imprimer les fichiers sources en langage Java, ou autres fichiers textes, avec une mise en page facilitant la lisibilité : bandeau indiquant le nom du fichier, numéros de page, mots-clés en gras, ... Taper `ccsi-print-source -h` pour son mode d'emploi.

4 Liste des projets

Pour chaque sujet de projet, nous faisons suivre entre parenthèses le nom de la personne prévue pour l'encadrer.

Cette personne représente votre « client ». Vous devez donc la consulter dès que vous avez des doutes sur le cahier des charges. Vous devez également confronter ce que vous voulez concevoir avec ce qu'elle attend.

Il est donc nécessaire de dialoguer régulièrement avec elle, en personne ou à défaut par téléphone ou par courrier électronique.

Indépendamment de la personne encadrant le projet, les deux enseignants du cours :

- **Norbert Kajler, bureau : L.012, Norbert.Kajler@mines-paristech.fr**
- **Fabien Moutarde, bureau : V.027, Fabien.Moutarde@mines-paristech.fr**

sont également disponibles pour répondre à vos questions techniques.

4.1 Log and Profile (N. Kajler) ★★

On souhaite permettre à des programmeurs Java d'ajouter à leur code de quoi conserver une trace de certaines actions effectuées à l'exécution, de chronométrer des parties du code, de visualiser sous forme d'animation certains éléments relatifs au programme en cours d'exécution, etc.

Dans sa version la plus simple, LAP pourra être utilisé comme suit, à savoir en ajoutant simplement des lignes telles que celles-ci à son code Java :

```
      ...
LAP.timerOn();
      ...
LAP.log("message");
      ...
LAP.timerOff();
      ...
LAP.log("la variable x vaut : " + x);
      ...
```

A l'exécution, chaque ligne `LAP.log()` produira l'affichage d'un message sur le flux de sortie standard. De même chaque ligne `LAP.timerOff()` affichera le temps écoulé depuis le déclenchement du timer par `LAP.timerOn()`.

Toutefois, il devra être possible d'instrumentaliser le code de manière beaucoup plus élaborée, sur le modèle suivant. Le programmeur commence par initialiser LAP en plaçant au début de son programme des lignes telles que :

```
LAPTimer t1 = new LAPTimer();
LAPTimer t2 = new LAPTimer();
LAPLog log1 = new LAPLog();
LAPLog log2 = new LAPLog();
LAPAlarm alarm1 = new LAPAlarm();
LAPRec recX = new LAPRec();
LAPRec recY = new LAPRec();
LAPRec recO = new LAPRec();
LAPinit();
```

Puis, il positionne librement dans son code des lignes telles que :

```
      ...
LAP.timerOn(t1);
LAP.timerOn(t2);
      ...
LAP.timerOff(t1);
      ...
LAP.log(log1, "message");
      ...
LAP.log(log2, "début du calcul ...");
LAP.log(log2, "NB : x et y sont enregistrés à chaque tour de boucle");
      ...
LAP.rec(recX, x);
LAP.rec(recY, y);
      ...
LAP.timerOff(t2);
      ...
if(x<0) LAP.ring(alarm1, "x=" + x);
      ...
LAP.rec(recO, obj);
LAP.timerOn(t2);
      ...
LAP.log(log1, "blablabla");
      ...
```

A l'exécution du programme, la ligne `LAP.init()` produit l'affichage d'une interface graphique permettant de contrôler dynamiquement l'exécution de la suite du programme, c'est-à-dire :

1. brancher ou débrancher chacun des outils déclarés par le programmeur (dans l'exemple ci-dessus il y a 8 outils : 2 timers, 2 logs, 1 alarm, et 3 recorders) ;
2. pour chaque outil, spécifier ses paramètres de fonctionnement, à savoir :
   - pour chaque log : affichage sur le flux de sortie standard ou dans la fenêtre de l'outil LAP ou dans un fichier à spécifier ; sélection des informations à afficher au début de chaque ligne parmi : thread, nom du fichier et ligne dans le code, date, heure ;
   - pour chaque alarme : affichage sur le flux de sortie standard ou affichage dans la fenêtre graphique ou dans un fichier à spécifier ou envoi d'un email à un destinataire à spécifier ;
   - pour chaque recorder : affichage en mode textuel sur le flux de sortie standard ou dans un fichier à spécifier ou dans la fenêtre de l'outil LAP ou alors affichage sous forme d'une courbe dans la fenêtre de l'outil LAP. En cas d'affichage sous forme de courbe, possibilité de préciser la couleur du tracé (les courbes pouvant s'afficher dans des zones séparées, ou bien se superposer, au choix de l'utilisateur) ;
3. sauvegarder la configuration LAP (ensemble des choix effectuées jusqu'ici) ;
4. charger une configuration LAP sauvegardée lors d'une exécution précédente ;
5. poursuite de l'exécution du programme avec mise en oeuvre des différents outils tels qu'ils ont été configurés.

La commande `LAP.rec()` devra pouvoir accepter comme deuxième argument soit un nombre (type : double) soit un objet quelconque à condition qu'il implante l'interface `LAPRecordable` (interface à définir avec l'idée que les objets "recordables" doivent pouvoir être affichées à la fois sous forme textuelle et sous forme de courbe).

Par ailleurs, il devra être possible de démarrer l'exécution automatiquement en spécifiant que la configuration LAP est à lire directement dans un fichier. Pour se faire, il devrait suffire d'invoquer `LAP.init()` avec en paramètre le nom du fichier de configuration à utiliser.

Enfin, le programme devra être conçu pour permettre à moindre coût son extension ultérieure (ajout de nouveaux types d'outils notamment).

4.2 Carrelages (N. Kajler) ★★★ à ★★★★

On se propose d'écrire un logiciel permettant de simuler le pavage d'un mur (ou d'un sol) à base de carreaux rectangulaires. L'objectif pour l'utilisateur de ce programme est de pouvoir rapidement construire et visualiser à l'écran un certain nombre de combinaisons avant de procéder à l'achat effectif, puis à la pose du carrelage.

On se propose donc de manipuler des *jeux de carreaux*. Un jeu de carreaux est constitué d'un ensemble de N *sortes* de carreaux de taille identique, conçus pour être combinés de manière plus ou moins libre. Chaque *sorte* de carreau possède sur sa face supérieure un « dessin » original au sein du jeu. Plus précisément, chaque sorte de carreau possède une combinaison de caractéristiques visuelles uniques au sein du jeu (couleur de fond, forme du dessin, couleurs du dessin, ...). Toutefois, pour ce qui nous concerne, une sorte de carreau sera caractérisée par une *liste de propriétés abstraites*, une *propriété* étant constituée d'un couple (nom de la propriété, valeur). En pratique le nombre et les noms des propriétés des N sortes de carreaux d'un jeu sera fourni par l'utilisateur.

Exemple : le jeu de carreau « ciel / oiseaux » se compose de cinq sortes de carreaux représentant soit le ciel bleu, soit un nuage, soit un oiseau en vol (en combinant 3 carreaux avec chacun une partie de l'oiseau, carreaux à poser impérativement côte à côte, dans le bon ordre, c-à-d. de gauche à droite en commençant par un carreau représentant une tête d'oiseau). Pour ce jeu (en fait *très* simple), on se contente de définir une seule propriété « `dessin` », de type énumérée, dont la valeur spécifie le dessin parmi : `ciel`, `nuage`, `tête`, `corps`, `queue`.

Note : on pourra considérer dans la suite que tous les carreaux sont en fait des carrés de même taille et possèdent une orientation unique (autrement dit on s'interdit toute rotation).

On veut écrire un programme permettant de paver une surface rectangulaire avec des carreaux issus d'un jeu donné en respectant un certain nombre de règles. Une règle est la spécification d'une contrainte d'interdiction ou d'obligation au niveau des propriétés des carreaux lors d'alignements horizontaux ou verticaux.

Exemple d'ensemble de règles compatibles avec le jeu « ciel / oiseaux » :
– toujours un corps d'oiseau à droite d'une tête d'oiseau ;
– toujours une queue d'oiseau à droite d'un corps d'oiseau ;
– toujours du ciel bleu autour des trois carreaux composant un oiseau ;
– toujours du ciel bleu autour d'un carreau comportant un nuage.

En termes plus formels, ces règles peuvent s'exprimer sous forme d'expressions telles que :
`TOUJOURS dessin=corps ADROITE dessin=tête.`
A priori, il devrait suffire de seulement quatre mots clés (`TOUJOURS`, `JAMAIS`, `ADROITE`, `AUDESSUS`) pour pouvoir exprimer la plupart des contraintes souhaitables en pratique. Au delà, rien n'empêche d'enrichir ce mini-langage par la suite pour permettre la spécification de règles plus complexes (telles que : « au minimum un oiseau par surface » ou « jamais d'oiseaux plus haut que les nuages »).

Étant donné un jeu de carreaux — par exemple 16 sortes de carreaux dont les dessins sont stockés, dans un même répertoire sous forme de fichiers au format GIF (ou autre format équivalent) — le programme devra permettre :

1. la définition des différentes propriétés (nom et type) associables à ce jeu de carreaux ;
2. pour chaque sorte de carreau du jeu, la constitution de sa liste de propriétés ;
3. la définition d'un (ou plusieurs) ensemble(s) de règles régissant l'assemblage des carreaux de ce jeu ;
4. la sauvegarde/lecture sur disque de ces données ;
5. le carrelage de surfaces rectangulaires une fois sélectionnés a) un jeu de carreaux, b) un ensemble de règles compatibles avec ce jeu, et c) la taille de la surface à carreler. Idéalement, l'utilisateur devrait pouvoir choisir entre deux modes : manuel ou automatique.
   En mode « manuel », l'utilisateur sélectionne un carreau sur une palette (représentant les différents carreaux du jeu) et le positionne à l'endroit de son choix sur la grille, l'ordinateur se contentant d'interdire la pose de carreaux aux emplacements incompatibles avec les règles de pavage.
   En mode « automatique », l'ordinateur remplit de manière aléatoire la grille tout en respectant les règles de pavage.

Facultatif (mais bien pratique) : sauvegarde/lecture des différents dessins ainsi créés ; possibilité de passer en mode automatique après placement manuel des premiers carreaux sur la grille ; possibilité de spécifier (sous forme de règles supplémentaires ?) la bonne manière de carreler les bords ; affichage à tout instant du nombre de carreaux de chaque sorte utilisés ; ...

4.3 Logique « floue » (F. Moutarde) ★★★ à ★★★★

Dans la logique classique, les variables sur lesquelles on raisonne sont booléennes, c'est-à-dire ne peuvent prendre que deux valeurs : vrai ou faux. Pourtant, nous utilisons très souvent dans la vie courante (et avec succès !) des raisonnements ou algorithmes manipulant des notions approximatives : par exemple, si notre douche n'est pas équipée d'un mitigeur thermostatique, nous sommes capables d'obtenir (et maintenir) une eau à une température convenable en appliquant une recette triviale du type : `si l'eau est` **`trop chaude,`** `alors augmenter le débit d'eau froide ; si elle est` **`trop froide,`** `alors augmenter le débit d'eau chaude`.
Dans cet exemple, on applique un raisonnement sur des valeurs qui *semblent* booléennes (`eau` **`trop chaude`**, `eau` **`trop froide`**), mais qui sont en réalité définies de façon ***floue***, puisqu'on ne saurait définir *précisément* la frontière entre `température` **`convenable`** et **`trop chaud`**.
La théories des « **sous-ensembles flous** » et la « **logique floue** » (inventées en 1965 par L. Zadeh) a justement pour but de définir des bases mathématiques permettant ce genre de raisonnement sur des quantités imprécises. L'objectif du projet est de développer un ensemble de classes permettant d'utiliser ce formalisme, et de les valider sur un ou plusieurs exemple(s) simple(s) d'application.

Les sous-ensembles flous Un sous-ensemble **flou** d'un ensemble E est défini par une « fonction d'appartenance » μ qui associe à tout élément e de E une valeur comprise entre 0 et 1 : son « degré d'appartenance ». Ainsi, alors que pour un sous-ensemble "classique" C on peut dire pour tout élément de E si oui ou non il appartient à C, dans le cas d'un sous-ensemble flou F tout élément de E peut lui appartenir "partiellement".

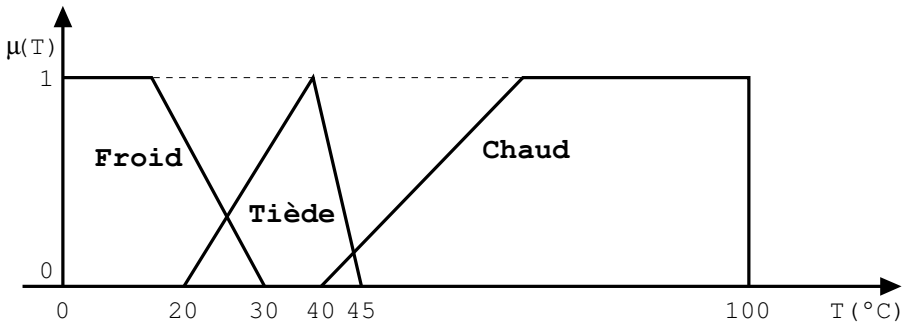

FIG. 1 – *Exemple de sous-ensembles flous :* ***Froid****,* ***Tiède*** *et* ***Chaud***

On étend aux sous-ensembles flous les opérations usuelles sur les sous-ensembles :

- complémentaire : si F est un sous-ensemble flou de fonction d'appartenance $\mu(F)$, alors son complémentaire $\overline{F}$ est défini par la fonction d'appartenance $\mu(\overline{F}) = 1 - \mu(F)$;
- intersection : si F et G sont deux sous-ensembles flous de fonctions d'appartenance $\mu(F)$ et $\mu(G)$, alors leur intersection $F \cap G$ est généralement définie par $\mu(F \cap G) = min(\mu(F), \mu(G))$ [2] ;
- union : $F \cup G$ est généralement défini par la fonction $\mu(F \cup G) = max(\mu(F), \mu(G))$ [3].

Dans le cadre de ce projet, on pourra se limiter à des sous-ensembles flous de type « élémentaire » et de forme triangulaire ou trapèzoïdale, et de type « complexe » obtenu par une combinaison des opérations décrites ci-dessus.

Quantification floue (ou « fuzzyfication ») Pour pouvoir utiliser la logique floue, il faut commencer par définir pour chaque variable intervenant dans le système une sorte de pseudo-partition de l'espace des valeurs possibles en plusieurs sous-ensembles flous se recouvrant plus ou moins et correspondant chacun à une "zone floue" nommée et ayant généralement un sens intuitif : ainsi la figure 1 ci-dessus montre un exemple de quantification de la variable `température` en trois sous-ensembles flous nommé respectivement **`Froid`**, **`Tiède`** et **`Chaud`**. La forme (généralement triangle ou trapèze) des fonctions d'appartenance des divers sous-ensembles flous de la pseudo-partition est moins importante que leur nombre et leurs positionnements respectifs.

Les règles floues La logique floue consiste à définir puis appliquer un ensemble de règles élémentaires du type `SI il fait` **`très froid,`** `ALORS chauffer` **`fort`** `(R1)`, où **`très froid`** correspond à un des sous-ensembles flous définis pour la variable `température`, et **`fort`** à un de ceux définis pour la variable `force_du_chauffage`. On se limite généralement à des implications (i.e., comme dans l'exemple `R1` ci-dessus, des règles de la forme `SI A, ALORS B` où `A` et `B` sont des « propositions »). La proposition `B`, qui constitue la « conclusion » de l'implication, est usuellement une « proposition simple » du type `telle`

2. Toutefois dans certains cas, l'intersection est plutôt définie par le *produit* des fonctions d'appartenance.
3. Toutefois dans certains cas, l'union est plutôt définie par la *moyenne arithmétique* des fonctions d'appartenance.

`variable doit appartenir à tel sous-ensemble flou` : par exemple dans la règle `R1`, la conclusion `chauffer` **`fort`** signifie que la variable `force_du_chauffage` doit appartenir au sous-ensemble flou **`fort`**. La proposition `A`, qui est la « prémisse » de l'implication, peut soit être également une « proposition simple », soit être une « proposition complexe » obtenue par une combinaison de propositions simples avec les « opérateurs booléens flous » `ET`, `OU`, ou `NON` (comme dans l'exemple de règle suivant : `SI la vitesse est` **`grande`** `ET la voiture devant est` **`très lente`**`, ALORS freiner` **`très fort`**).

Le raisonnement flou Le principe essentiel du raisonnement flou est le suivant : plus la prémisse est vraie, plus la conclusion doit l'être aussi. Ainsi, si la prémisse de la règle `R1` est vraie *à 60%* (*i.e.* si le degré d'appartenance de la `température` au sous-ensemble flou **`très froid`** est de 60%), alors la `force_du_chauffage` devra appartenir au sous-ensemble flou obtenu par le plafonnement à 60% du sous-ensemble **`fort`**. Dans le cas où la prémisse `A` est complexe (par exemple de type `A1 ET A2`), son « degré de vérité » sera calculé par la formule définissant l'opération correspondante sur les sous-ensembles flous (cf. plus haut) : le `NON` correspondant au complémentaire, le `ET` à l'intersection, et le `OU` à l'union (ainsi, par exemple, le « degré de vérité » de `A1 ET A2` est généralement pris égal au *minimum* de ceux de `A1` et de `A2`).

Composition des règles et « défuzzyfication » Le raisonnement flou produit donc dans un premier temps une prescription sous forme de sous-ensemble flou. En fait, il produit même généralement une *combinaison* de **plusieurs** sous-ensembles flous (affecté chacun d'un degré de validité), car il y a normalement plus d'une règle, et il arrive souvent que les prémisses de *plusieurs* règles aient simultanément un degré de vérité supérieur à 0. Mais au final, il faut bien obtenir une valeur précise pour les variables calculées (celles servant à commander le système dans le cas d'une application de type « commande floue »). Il faut donc convertir une combinaison de plusieurs sous-ensembles flous en une valeur unique. Il existe plusieurs variantes pour cette étape cruciale de la logique floue (nommée « défuzzyfication »). Les trois principales sont : la technique du maximum, la technique de la moyenne pondérée, et enfin la technique du centre de gravité (voir figure 2) qui est la plus coûteuse en calculs mais donne souvent les meilleurs résultats (voir les références pour plus de détails).

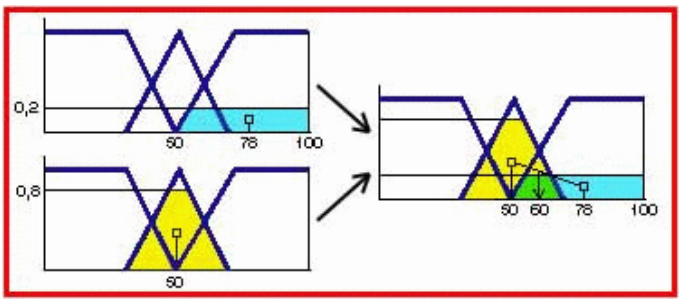

FIG. 2 – *Exemple de « défuzzyfication ». On suppose qu'une règle ayant comme conclusion le sous-ensemble flou le plus à droite est vraie à 20% (cf. graphique en haut à gauche), et qu'une seconde règle ayant comme conclusion le sous-ensemble flou du milieu est vraie à 80% (cf. graphique en bas à gauche) ; alors, la prescription obtenue par la technique du centre de gravité est la valeur 60, qui est l'abscisse du centre de gravité de la surface correspondant à l'union des deux sous-ensembles flous "conclusions" (cf. graphique de droite).*

Exemples simples d'application Afin de valider la librairie de classes, on réalisera une ou plusieurs mini-application(s) de commande floue. On pourra *par exemple* choisir :

1. une commande de freinage ultra-simpliste tel que celui décrit sur la page `http://philduweb.free.fr/contributions/fuzzy/frein.htm` ;
2. une régulation conjointe de température et de débit de l'eau d'une douche (cf. ci-dessous) ;
3. toute autre idée (à proposer dans l'avant-projet).

Pour la *suggestion* numéro 2 (régulation de douche), on pourra supposer que sont mesurées la température et le débit *total*, et que l'on peut augmenter ou réduire les ouvertures d'arrivée d'eau chaude et froide (comme avec deux robinets séparés) ; on se donnera une loi *non-linéaire* (du type de la figure 3 ci-dessous) déterminant le débit *nominal* d'un robinet en fonction de son ouverture ; en outre, pour une ouverture donnée du robinet d'eau chaude (resp. froide) le débit *effectif* d'eau chaude (resp. froide) fluctuera aléatoirement autour de sa valeur nominale pour représenter les perturbations extérieures dues aux autres utilisateurs (par exemple $debitEffectif(t) = debitNominal(t) \times fluctuation(t)$ avec $fluctuation(t)$ tiré pour chaque t de manière aléatoire uniforme dans l'intervalle $[0,8 ; 1,2]$).

Pour la définition des quantifications et règles floues, on pourra *par exemple* tester (et améliorer !!) le principe (purement empririque et trivial) résumé dans le tableau ci-dessous :

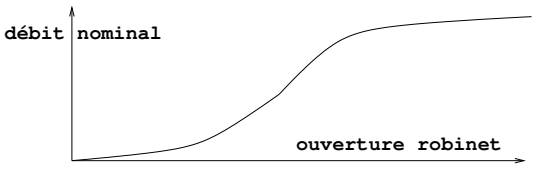

FIG. 3 – *Exemple possible de dépendance du débit* **nominal** *en fonction de l'ouverture du robinet.*

débit	**Faible**	**OK**	**Fort**
température			
Froid	**Plus** d'EC	**Plus** d'EC , **Moins** d'EF	**Moins** d'EF
OK	**Plus** d'EC , **Plus** d'EF	-	**Moins** d'EC , **Moins** d'EF
Chaud	**Plus** d'EF	**Plus** d'EF, **Moins** d'EC	**Moins** d'EC

où EC et EF signifient respectivement Eau Chaude et Eau Froide, et où **Plus** et **Moins** correspondent à des sous-ensembles flous pour la variation d'ouverture de chaque robinet.

Références Pour en savoir plus sur la logique floue, on pourra par exemple visiter les sites :

- http://fr.wikipedia.org/wiki/Logique_floue
- http://elap.montefiore.ulg.ac.be/elap/documents/fuzzy/
- http://perso.club-internet.fr/bmantel/pages/logfloue/logfloue00.html

et se référer aux livres (présents à la bibliothèque de l'Ecole) de la spécialiste française de la logique floue, Bernadette Bouchon-Meunier :

- La logique floue et ses applications, B.Bouchon-Meunier, Addison-Wesley, 1995.
- La logique floue, B.Bouchon-Meunier , Que sais-je (Presses Universitaires de France), 1994.

4.4 Support Vector Machines (SVM) (F. Moutarde) ★★ à ★★★

Les "Support Vector Machines" (SVM, traduit en "Séparateurs à Vaste Marge") sont un type de classifieur et un algorithme d'apprentissage statistique très utilisé, notamment dans le domaine de la reconnaissance de forme.

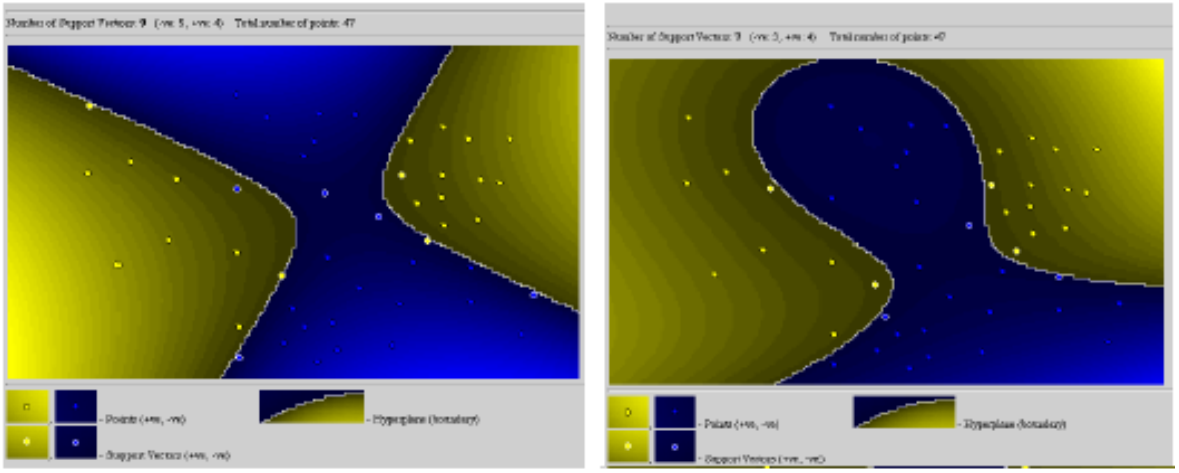

FIG. 4 – *Illustration de classifieur SVM sur un exemple-jouet de classification dans* R^2.

Une SVM se caractérise par un type de "noyau" ("kernel"), qui correspond au produit scalaire utilisé $k(x,z)$, et définit indirectement une transformation de l'espace des entrées vers un espace de + grande dimension dans lequel une séparation linéaire optimale sera calculée. Ce noyau est le plus souvent choisi parmi les plus courants listés ci-dessous :

- polynomial $k(x,z) = (x.z+1)^q$
- gaussien (aussi nommé RBF) $k(x,z) = e^{-\frac{|x-z|^2}{2*\sigma^2}}$

L'algorithme d'apprentissage consiste, dans l'espace des $\Phi(x)$ (où la transformation Φ est définie par $\Phi(x).\Phi(z) = k(x,z)$), à trouver la frontière linéaire (hyperplan) séparant correctement les 2 classes tout en "maximisant la marge" entre les exemples et cette frontière. Il s'agit donc d'une maximisation sous contrainte, qui peut se résoudre par la méthode de Lagrange, laquelle ramène à un problème quadratique.

Au final, le classifieur obtenu est de la forme : $h(x) = \sum_k (\alpha_k * k(x_k, x))$ où les x_k sont les "vecteurs de support" qui forment un sous-ensemble (normalement de petit cardinal) des exemples d'apprentissage.

Le but du projet est d'implémenter les SVM en Java, avec la contrainte de s'insérer dans un framework existant et intégrant déjà divers autres algos d'apprentissage de classifieurs (rétropropagation pour les réseaux neuronaux à couches, adaBoost, ...) pour les bases d'images. Un apercu de cette librairie existante dans laquelle il s'agit de s'insérer est donné Fig. 5.

Des explications et documents complémentaires sur les SVMs seront fournis, ainsi que le jar de la librairie existante, dont un "Reverse" sous Modelio pourra être fait afin d'en obtenir un modèle UML. Des exemples de base d'images à apprendre seront également mis à disposition, afin de tester le bon fonctionnement de l'implémentation des SVMs, ainsi que la bonne intégration dans l'outil jLEVIS.

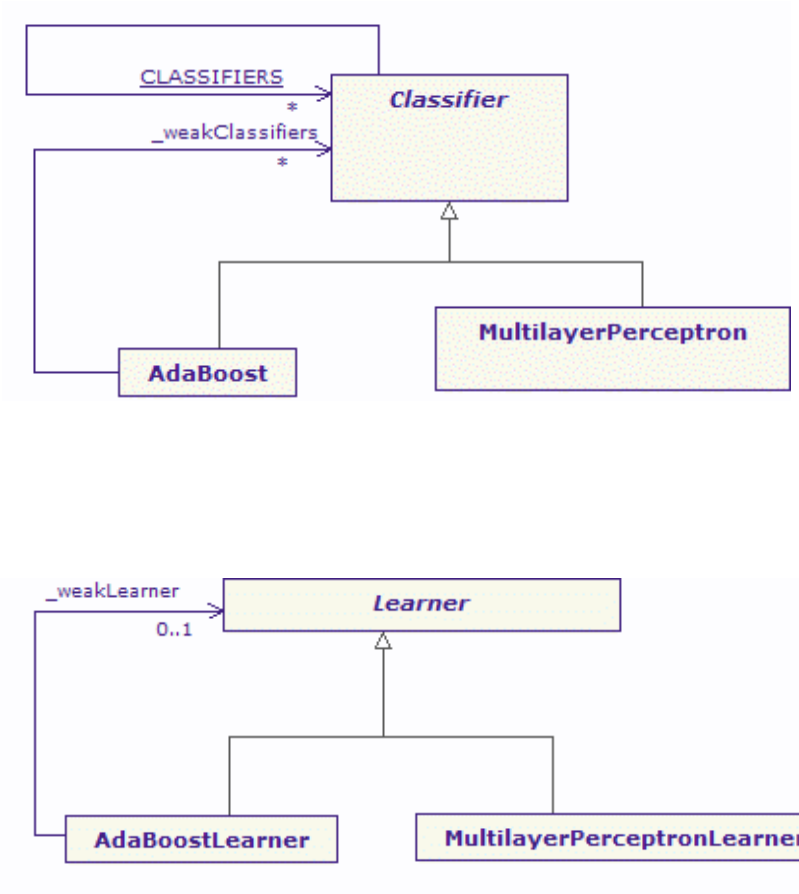

FIG. 5 – *La librairie existante dans laquelle le développement des SVMs doit s'insérer.*